J. Functional Programming **2** (3): 345–364, July 1992 © 1992 Cambridge University Press

Efficient self-interpretation in lambda calculus

TORBEN Æ. MOGENSEN
DIKU, University of Copenhagen, Denmark

Abstract

We start by giving a compact representation schema for λ-terms, and show how this leads to an exceedingly small and elegant self-interpreter. We then define the notion of a *self-reducer*, and show how this too can be written as a small λ-term. Both the self-interpreter and the self-reducer are proved correct. We finally give a constructive proof for the second fixed point theorem for the representation schema. All the constructions have been implemented on a computer, and experiments verify their correctness. Timings show that the self-interpreter and self-reducer are quite efficient, being about 35 and 50 times slower than direct execution using a call-by-need reductions strategy

Capsule review

In this paper the author introduces an internal coding for lambda terms that assigns to $M \in \Lambda$ a normal for $\lceil M \rceil$. The coding makes use of the representation of data types in second order lambda calculus and hence in type-free lambda calculus. A self-interpretation E and a self reducer R are constructed in a very elegant way. These satisfy for all $M \in \Lambda^0$.

$$\begin{aligned} E \lceil M \rceil &=_\beta M; \\ R \lceil M \rceil &=_\beta \lceil M^{nf} \rceil, && \text{if } M \text{ has } nf\ M^{nf}; \\ &\text{has no } nf, && \text{else.} \end{aligned}$$

The first equation is even valid for all $M \in \Lambda$.

1 Preliminaries

The set of λ-terms Λ is defined by the abstract syntax

$$\Lambda = V \mid \Lambda\Lambda \mid \lambda V.\Lambda$$

where V is a countable infinite set of distinct variables. (Possibly subscripted) lower case letters $a, b, x, y, \ldots$, are used for variables, and capital letters $M, N, E, \ldots$ for λ-terms. We will assume familiarity with the rules for reduction in the lambda calculus, and mention these without reference. The shorthand $\lambda x_1 \ldots x_n . M$ abbreviates $\lambda x_1 \ldots \lambda x_n . M$ and $M_1 M_2 \ldots M_n$ abbreviates $(\ldots(M_1 M_2) \ldots M_n)$.

Two λ-terms are considered *identical* if they only differ in the names of bound variables (i.e. they are α-convertible), and *equal* if they can be β-reduced to identical λ-terms. We use the symbol $\equiv$ for identity and $=_\beta$ for equality. $M \to N$ means that M reduces to N by one β-reduction. $M \twoheadrightarrow N$ means that M reduces to N by zero or more β-reductions.

When a λ-term M has a normal form, we use the notation NF_M to mean the normal form of M. NF_Λ is the set of λ-terms in normal form. $\mathsf{FV}(M)$ denotes the set of free variables in M. Λ^0 denotes the set of closed λ-terms.

Data is usually represented in pure lambda calculus by λ-terms in normal form (e.g. Church numerals). Normal form λ-terms are indeed 'constants' with respect to reduction, which is a natural requirement for data. Normally, data values are represented in such a way that the required operations on them are simple to do in pure lambda calculus. Note that the only way to inspect a λ-term from within the lambda calculus is to apply it to one or more arguments. Also, note that an application in itself does not involve evaluation, it is a mere syntactic construction. Reduction must be explicitly stated by NF or by use of the $=_\beta$ relation.

A *representation schema* for the lambda calculus is an injective (up to identity) mapping $\lceil\cdot\rceil : \Lambda \to \mathsf{NF}_\Lambda$. That is, $\lceil\cdot\rceil$ will represent any λ-term by a λ-term in normal form. Furthermore, the representations of two λ-terms are identical *iff* the λ-terms are.

A *self-interpreter* is a λ-term E, such that

$$E\lceil M\rceil =_\beta M$$

for any λ-term M. That is, E applied to the representation of M is equal to M itself.

A *self-reducer* is a λ-term R, such that

$$R\lceil M\rceil =_\beta \lceil \mathsf{NF}_M\rceil$$

for any λ-term M that has a normal form. If M has no normal form, neither should $R\lceil M\rceil$.

Thus R applied to the representation of M reduces to the representation of M's normal form, *iff* such exist.

The difference between a self-interpreter and a self-reducer is mainly that the self-reducer reduces to a representation of the normal form of its argument, whereas the self-interpreter reduces directly to the normal form itself, if such exist. Note that β-convertibility is required between a λ-term M, and the self-interpreter applied to M, even when no normal form exist for M. The right-hand side of the equation involving the self-reducer is undefined if M has no normal form, so we need to specify separately what should happen in that case.

The second fixed point theorem (see, for example, Barendregt, 1984) states that there for all λ-terms F exist a λ-term X such that

$$F\lceil X\rceil =_\beta X.$$

Furthermore, there exists a 'second-fixed-point' combinator Θ, such that

$$X \equiv \Theta\lceil F\rceil \Rightarrow F\lceil X\rceil =_\beta X.$$

Note that this theorem is dependent on the representation schema, so we need to prove that it is valid for the schema we present below.

2 A representation schema

Barendregt (1991) used a two-step representation schema: first λ-terms were represented as Gödel numbers, then these were represented as Church numerals. While this is a theoretically nice encoding, as it combines two well-known constructions, it is hardly compact. The size of representations of λ-terms using this schema grows (at least) exponentially in the size of the terms. Operations on this representation are also extremely expensive.

To construct a more compact representation schema we will use a combination of *higher order abstract syntax* (Pfenning and Elliot, 1988) and a well-known way of representing signatures using λ-terms. The latter is, in fact, so well-known that the author doesn't know where it originated, but it is used in Reynolds (1985). Steensgaard-Madsen (1989) contains this, and several other ways of representing data in pure lambda calculus.

Higher order abstract syntax is an abstract syntax representation that extends syntax trees with the abstraction mechanism of the lambda calculus. The idea is to represent scope rules by λ-abstraction. It is no surprise, then, that higher order abstract syntax easily captures the scope rules of the lambda calculus. A higher order abstract syntax representation $\lfloor \cdot \rfloor$ of Λ using unary constructors *Var* and *Abs* and a binary constructor *App* is

$$\begin{aligned} \lfloor x \rfloor &= Var(x) \\ \lfloor M\,N \rfloor &= App(\lfloor M \rfloor, \lfloor N \rfloor) \\ \lfloor \lambda x\,.\,M \rfloor &= Abs(\lambda x\,.\,\lfloor M \rfloor). \end{aligned}$$

Note that binding of variables is handled meta-circularly by binding of variables. For example, the coding of $\lambda x\,.\,x\,x$ is

$$Abs(\lambda x\,.\,App(Var(x), Var(x))).$$

Representation of signatures can be done by combining and generalizing the standard representations of pairs and booleans in the pure lambda calculus.

Given a signature Σ, for any sort S in Σ let Sc_i, $i = 1 \ldots n_s$ be the constructors in that sort. We represent the term $Sc_i(t_1, \ldots, t_m)$ by

$$\lambda x_1, \ldots, x_{n_s}\,.\,x_i\,\overline{t_1}, \ldots, \overline{t_m}$$

where $\overline{t_j}$ is the representation of the term t_j. As an example, the signature of lists can be represented as

$$\begin{aligned} \overline{Nil} &\equiv \lambda xy\,.\,x \\ \overline{Cons(A,B)} &\equiv \lambda xy\,.\,y\bar{A}\bar{B}. \end{aligned}$$

Note that pairs are isomorphic to a sort with one dyadic constructor, *Pair*. The representation of this is $Pair(A, B) \equiv \lambda x\,.\,x\,A\,B$, which is the traditional representation of pairs. Booleans are isomorphic to a signature with two nullary constructors *True* and *False*. These are represented by $True \equiv \lambda xy\,.\,x$ and $False \equiv \lambda xy\,.\,y$. Again, this is the standard representation.

This coding allows a switch (*case*) on constructor names to be implemented in a simple fashion in the lambda calculus. For example

$$\overline{case\ E\ of\ Nil \Rightarrow F;\ Cons(a,b) \Rightarrow G} \equiv \bar{E}\bar{F}(\lambda a\,b\,.\,\bar{G}).$$

When applied to booleans, this construction yields the classical implementation of *if-then-else* in lambda calculus.

Combining higher order abstract syntax and the coding of signatures, we get the following representation schema for the lambda calculus

$$\begin{aligned}\lceil x \rceil &\equiv \lambda abc.a\,x\\ \lceil M\,N \rceil &\equiv \lambda abc.b\lceil M \rceil\lceil N \rceil\\ \lceil \lambda x.M \rceil &\equiv \lambda abc.c(\lambda x.\lceil M \rceil)\end{aligned}$$

where a, b, c are variables *not* occurring free in the λ-term on the left-hand side of the equation. Such variables can always be found, for example, by choosing from the start three variables that do not occur in the entire λ-term. It is clear that the conditions (normal form and injectivity) for representation schemae are fulfilled. Note that

$$\mathsf{FV}(M) = \mathsf{FV}(\lceil M \rceil).$$

As an example, the coding of $\lambda x.x\,x$ is shown below:

$$\lambda abc.\,c(\lambda x.\,(\lambda abc.\,b(\lambda abc.\,a\,x)\,(\lambda abc,a.\,x))).$$

It is easy to see that this representation is linear in the size of the represented λ-terms. In fact, the size (measured as the number of variables plus the number of applications plus the number of abstractions) of the representation is roughly seven times the size of the original term. Operations like testing, decomposition and building of terms are quite efficient using this representation, requiring only a few β-reductions each.

Useful variants of this representation schema exist; one can, for example, avoid constructors on bound variables at the cost of complicating the self-interpreter slightly. The presented schema is, in our opinion, the simplest and most elegant.

Pfenning and Lee (1991) use a similar representation for their (almost) metacircular interpreter for typed lambda calculus. Their representation schema is somewhat more complex and even after removal of the complications that handle type parameters, the result is not quite the same.

3 Self-interpretation

We will (for the sake of readability) initially present the self-interpreter using recursive equations and uncoded higher order syntax. Then we will use the coding from above to convert this into the pure lambda calculus.

β-reduction of the abstractions in the higher order syntax is used to perform substitution in the interpreter. Thus no environment is needed. The effect is that some β-redexes perform substitution, and others simulate reduction in the interpreted program. Apart from this slight subtlety, the interpreter below is remarkably easy to understand

$$\begin{aligned}E[\mathit{Var}(x)] &= x\\ E[\mathit{App}(M,N)] &= E[M]\,E[N]\\ E[\mathit{Abs}(M)] &= \lambda v.\,E[(M\,v)].\end{aligned}$$

It is easy to prove by induction that recursive application of these equations and reduction of the redexes of form $(M\,v)$ from the third equation, will reduce $E[\lceil N \rceil]$ to a λ-term that is *identical* to N, for any λ-term N. Note that this includes λ-terms with free variables. This is not the case for the representation that Barendregt used, as explained in his article.

We now code the syntax as λ-terms, replace the pattern matching by application (as explained in the previous section), and use a fixed point combinator to eliminate explicit recursion. This yields the complete self-interpreter

$$\begin{aligned} E \equiv Y\lambda e.\lambda m.m\,(\lambda x.x) \\ (\lambda mn.(e\,m)\,(e\,n)) \\ (\lambda m.\lambda v.e(m\,v)) \end{aligned}$$

where $$Y \equiv \lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x)).$$

By a similar reasoning as above, it can be shown that it is possible to reduce $E\lceil N\rceil$ to a λ-term that is identical to N. This is stronger than our requirement for self-interpretation: we didn't require reduction of $E\lceil N\rceil$ *to* N, just that $E\lceil N\rceil$ and N could be reduced to identical terms. Appendix A contains the proof of correctness for E.

4 Self-reduction

Reduction to normal form consists of repeatedly selecting and reducing redexes until none are left. The selection is usually made explicit in a reducer by some programmed strategy, but perhaps we can let this be implicit in the underlying reduction of the self-reducer applied to its argument? This has the advantage that whatever sharing of reductions that the system makes might be exploited by the self-reducer to get a similar sharing.

We *do* present such a self-reducer. It is a bit more complex than the self-interpreter, but, we believe, less complex that a self-reducer with explicit redex selection.

The benefits of returning a (representation of) a normal form rather than a function when evaluating lambda terms are discussed by Berger and Schwichtenberg (1991). To obtain this they define a procedure that, given the function returned by the evaluator, returns a normal form term corresponding to the function. The procedure is controlled by the type of the term in question, so it is not applicable to our problem. The idea of letting the underlying evaluation mechanism handle the reduction is, however, similar to ours.

The reducer is restricted to closed λ-terms. We believe it is not possible to make a reducer that handles general λ-terms using the representation schema presented above. This has to do with the fact that it is not possible to distinguish free and bound variables, and these have to be treated differently. This is in contrast to what happened in the interpreter, where they are indeed treated in the same fashion. If the representation schema is extended to distinguish free and bound variables, it would be a simple matter to extend the self-reducer below to handle terms with free variables correctly.

The reducer works in a rather novel way. The method avoids repeated rewrites and scans for redexes in the term, thus (as we shall see) making it quite efficient. For rather subtle reasons having to do with the higher order abstract syntax (you cannot inspect the bindings of variables), it is difficult to write a reducer in a more straightforward manner using the present representation schema.

When we have an application $App(M, N)$, we want to evaluate or reduce M to a function that takes (the normal form of) N as argument (as in the interpreter). But what if M does not reduce to an abstraction? This could happen when reducing the body of a top-level λ-abstraction, for example, in $\lambda x . x E$. Here x must evaluate to a function that takes (the normal form of) E and returns the representation of the expression $x\,\mathsf{NF}(E)$, which in turn becomes part of the representation of $\lambda x . x\,\mathsf{NF}(E)$. The result of applying a variable to an argument could again be applied to a further argument, etc., so any expression that does not reduce to an abstraction must evaluate to a function that builds an application node when given an argument.

When an expression is applied to an argument, we want it to be a function, and when it is not, we want it to be a representation. These conflicting demands can be handled using an evaluation function in the first case, and a reduction function in the latter, or (as we will do) a single function which produces a *pair* of a function and a representation.

Again, we start by showing an equational system using higher order abstract syntax. We have used the constructor *Pair* and selectors *fst* and *snd*

$$\begin{aligned}
R[M] &= snd(R'[M]) \\
R'[Var(x)] &= x \\
R'[App(M, N)] &= fst(R'[M])\, R'[N] \\
R'[Abs(M)] &= let\ g = \lambda v . R'[(M\ v)]\ in \\
&\qquad\qquad Pair(g, Abs(\lambda w . snd(g P[Var(w)]))) \\
P[M] &= Pair(\lambda v . P[App(M, snd(v))], M).
\end{aligned}$$

$R'[M]$ produces a pair of a function and a representation, so we take the second component to get the representation for the final answer. Variables will be substituted by such pairs by the time we see them, so the pair is just returned. If we have an application $App(M, N)$, we evaluate M to a pair and apply the first component to the (pair) value of N. If we have an abstraction, we build a pair of a function and a representation. The function behaves like the functional value in the interpreter. The representation is an abstraction node with a body that is obtained by evaluating the body of the original abstraction to a pair, and then taking the second component. When evaluating the body, the bound variable is given a value that builds an application node whenever it is applied to an argument. This is handled by the function P. Note that the recursive nature of P ensures that any number of arguments produces a corresponding number of application nodes. If, for example, the variable x is applied to an argument M, an application $x\,M$ is produced by applying the function part of the pair bound to x to M. But if $x\,M$ is again applied to an argument N, another application $(x\,M\,N)$ must be produced, etc. This is handled by using P on the constructed applications. If P was unfolded fully, it would yield an infinite term, but unless no normal form exists for the argument to the reducer, only a finite part of the result of P will be needed. The application of the equations and reduction of redexes can (but need not) be driven by having to reduce the *snd* operation against a *Pair*. In the rule for abstractions, this will introduce another application of *snd*, which must also be reduced, etc. If *snd* is reduced against the *Pair* from the *P* rule, no extra reduction is required.

An interesting thing to note is that there is no *required* reduction strategy: it depends on how the equational system is used.

The representations of *Pair*, *fst* and *snd* is just a special case of the representation of signatures described above

$$\begin{aligned} Pair(A,B) &\equiv \lambda x.x\,A\,B \\ fst(A) &\equiv A(\lambda ab.a) \\ snd(A) &\equiv A(\lambda ab.b). \end{aligned}$$

So we get the following complete self-reducer

$$R \equiv \lambda m.R'm(\lambda ab.b)$$

where
$$\begin{aligned} R' \equiv Y\lambda r.\lambda m.m\,&(\lambda x.x) \\ &(\lambda mn.(r\,m)(\lambda ab.a)(r\,n)) \\ &(\lambda m. \\ &\quad(\lambda g. \\ &\qquad\lambda x.x\,g(\lambda abc.c(\lambda w.g(P\,\lambda abc.a\,w)(\lambda ab.b)))) \\ &\quad\lambda v.r(m\,v)) \end{aligned}$$

where $P \equiv Y\lambda p.\lambda m.(\lambda x.x(\lambda v.p(\lambda abc.b\,m(v(\lambda ab.b))))\,m)$

and $Y \equiv \lambda h.(\lambda x.h(x\,x))(\lambda x.h(x\,x))$.

Though this might seem overwhelming, it is a fairly mechanical translation of the equational system from above. In Appendix B we present an outline of a proof of correctness for the self-reducer. It is in two parts: the first part shows that $R\lceil M\rceil =_\beta \lceil \mathsf{NF}_M\rceil$ when M has a normal form NF_M. The other part shows that $R\lceil M\rceil$ has no normal form if M doesn't. The first part relies on two basic properties of R: if $M =_\beta N$, then $R\lceil M\rceil =_\beta R\lceil N\rceil$, and if M is in normal form then $R\lceil M\rceil =_\beta \lceil M\rceil$. The second part is more complex and proves that if there exist an infinite left-most reduction path for M, then there exist an infinite quasi-left-most reduction path for $R\lceil M\rceil$.

5 Benchmarks

Both the self-interpreter and the self-reducer have been tried out, running on a call-by-need lambda calculus reducer implemented (using a 'pair-of-value-and-representation' structure similar to the self-reducer) in Chez Scheme (Dybvig, 1987) on a SPARC-station. The experiments indicate that both 'programs' are correct with

Table 1. *Reduction times*

Reduction	Time (s)	Normalized
$(Ackermann\ 3_{Church}) \twoheadrightarrow 61_{Church}$	0·77	1
$E\ \lceil(Ackermann\ 3_{Church})\rceil \twoheadrightarrow 61_{Church}$	26·59	35
$R\ \lceil(Ackermann\ 3_{Church})\rceil \twoheadrightarrow \lceil 61_{Church}\rceil$	38·35	50

respect to the equations. In Table 1 are shown timings for reducing ($Ackermann\ 3_{Church}$) to normal form, where $Ackermann$ is Ackermann's function on Church numerals

$$Ackermann \equiv \lambda n . n\ (\lambda f m . f(m f(\lambda x . x)))\ (\lambda m f x . f(m f x))\ n$$

and

$$3_{Church} \equiv \lambda f x . f(f(f x)).$$

The timings are shown both in seconds and relative to direct execution. They show the reduction performed directly, then using the self-interpreter and finally using the self-reducer. The slowdown of the interpreter over direct execution is comparable to the usual difference between interpreted and compiled languages. Also, even though it returns a representation, the reducer is not much slower than the interpreter. Note that $E\ulcorner M\urcorner$ could first be reduced to M, and then to normal form. Such a strategy would make the required number of reductions much smaller than when call-by-need reduction is used. This would correspond closely to partial evaluation of the interpreter with respect to $\ulcorner M\urcorner$ followed by execution of the residual program. (See Gomard and Jones, 1991 for more details.) A forthcoming paper will describe self-applicable partial evaluation of pure lambda-calculus using the same techniques as in this paper.

6 The second fixed-point theorem

The second fixed-point theorem shows the existence and construction of λ-terms with special properties, like a self-representing term. This corresponds to the usual puzzle of writing a program that returns its own 'text'. The theorem is due to Kleene (1952), and Barendregt (1984) shows a version for the lambda calculus. Since the theorem involves representation of terms, it is necessary to prove if for any particular representation schema. It is fortunately easy to do so.

The second fixed point theorem states that there for all λ-terms F exist a λ-term X such that

$$F\ulcorner X\urcorner =_\beta X.$$

It is even the case that

$$X \twoheadrightarrow F\ulcorner X\urcorner.$$

Furthermore, there exists a 'second-fixed-point' combinator Θ, such that

$$X \equiv \Theta\ulcorner F\urcorner \Rightarrow X \twoheadrightarrow F\ulcorner X\urcorner.$$

We start by assuming that we have a λ-term Q, such that

$$Q\ulcorner M\urcorner \twoheadrightarrow \ulcorner\ulcorner M\urcorner\urcorner.$$

That is, Q adds a level of representation to an already represented λ-term. Note that the restriction that Q is applied only to represented arguments is necessary, as explained in Barendregt (1991). Now, define

$$X \equiv A\ulcorner A\urcorner$$

where

$$A \equiv \lambda n . F\, App(n(Q n)).$$

Now we can see that

$$\begin{array}{lll} X \equiv A\lceil A\rceil & & \\ \quad \rightarrow F\,App(\lceil A\rceil(Q\lceil A\rceil)) & & \text{by definition of } A \\ \quad \twoheadrightarrow F\,App(\lceil A\rceil(\lceil\lceil A\rceil\rceil)) & & \text{by property of } Q \\ \quad \equiv F\lceil(A\lceil A\rceil)\rceil & & \text{by representation of applications} \\ \quad \equiv F\lceil X\rceil & & \text{by definition of } X. \end{array}$$

So X is indeed a fixed point of F. Note that if F is the identity, X is a self-representing term. All we need now is to construct Q. First we show Q as an equational system, and then as a λ-term

$$\begin{array}{ll} Q\lceil Var(x)\rceil & = Abs(\lambda a.\,Abs(\lambda b.\,Abs(\lambda c.\,App(Var(a), Var(x))))) \\ Q\lceil App(M,N)\rceil & = Abs(\lambda a.\,Abs(\lambda b.\,Abs(\lambda c.\,App(App(Var(b), Q\lceil M\rceil), Q\lceil N\rceil)))) \\ Q\lceil Abs(M)\rceil & = Abs(\lambda a.\,Abs(\lambda b.\,Abs(\lambda c.\,App(Var(c), Abs(\lambda v.\,Q\lceil(M\,v)\rceil))))). \end{array}$$

The idea is simple: we represent the λ-terms that correspond to the constructors, and put the representations of the arguments inside these. The lambda calculus version of Q is

$$\begin{array}{l} Q \equiv Y\lambda q.\lambda m.m\,(\lambda x.(\lambda abc. \\ \qquad\qquad c(\lambda a'.(\lambda abc. \\ \qquad\qquad\quad c(\lambda b'.(\lambda abc. \\ \qquad\qquad\qquad c(\lambda c'.(\lambda abc. \\ \qquad\qquad\qquad\quad b(\lambda abc.a\,a')\,(\lambda abc.a\,x))))))))) \\ \qquad\quad (\lambda mn.(\lambda abc. \\ \qquad\qquad c(\lambda a'.(\lambda abc. \\ \qquad\qquad\quad c(\lambda b'.(\lambda abc. \\ \qquad\qquad\qquad c(\lambda c'.(\lambda abc. \\ \qquad\qquad\qquad\quad b(\lambda abc.b(\lambda abc.a\,b')\,(q\,m))\,(q\,n))))))))) \\ \qquad\quad (\lambda m.(\lambda abc. \\ \qquad\qquad c(\lambda a'.(\lambda abc. \\ \qquad\qquad\quad c(\lambda b'.(\lambda abc. \\ \qquad\qquad\qquad c(\lambda c'.(\lambda abc. \\ \qquad\qquad\qquad\quad b(\lambda abc.a\,c') \\ \qquad\qquad\qquad\qquad (\lambda abc.c(\lambda v.q(m\,v))))))))))), \end{array}$$

where $\quad Y \equiv \lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x)).$

The proof that Q has the stated property is simple, but takes some space, so we leave it as an exercise for the reader.

We have constructed a self-representing term by using the construction in the theorem. Instead of letting F be $\lambda x.x$, we simply omit it. We will not present the self-reducing term in its full form, as it is rather large. It has been reduced by computer, and it does indeed yield its own representation.

The construction of a second-fixed-point combinator can be derived in a few steps from the requirement

$$\Theta\lceil F\rceil \twoheadrightarrow F\lceil\Theta\lceil F\rceil\rceil.$$

Using the self-interpreter E, we see that if

$$\Theta \twoheadrightarrow \lambda f.(E f)\, App(\lceil\Theta\rceil, Q f)$$

then Θ will be a second-fixed-point combinator. The proof is simple:

$$\begin{aligned}
\Theta\lceil F\rceil &\twoheadrightarrow (\lambda f.(E f)\, App(\lceil\Theta\rceil, Q f))\lceil F\rceil && \text{by assumption}\\
&\rightarrow (E\lceil F\rceil)\, App(\lceil\Theta\rceil, Q\lceil F\rceil)\\
&\twoheadrightarrow F\, App(\lceil\Theta\rceil, Q\lceil F\rceil) && \text{by property of } E\\
&\twoheadrightarrow F\, App(\lceil\Theta\rceil, \lceil\lceil F\rceil\rceil) && \text{by property of } Q\\
&\equiv F\lceil\Theta\lceil F\rceil\rceil && \text{by representation of applications.}
\end{aligned}$$

But this property of Θ is just a special case of the second fixed-point theorem. Thus

$$\Theta \equiv A\lceil A\rceil$$

where
$$A \equiv \lambda n.G\, App(n(Q\, n))$$
$$G \equiv \lambda t.\lambda f.(E f)\, App(t, Q f)$$

Θ is quite large when written as a pure λ-term, so we will refrain from doing so. Doubtless there exist much smaller second-fixed-point combinators. Indeed, it is easy to reduce Θ somewhat by unfolding the application of G in the definition of A, and by abstracting out common subexpressions in Q.

Appendix A: Proof of correctness for the self-interpreter

We will prove

Theorem 1

$$E\lceil M\rceil \twoheadrightarrow M$$

where E is the self-interpreter:

$$E \equiv Y E'$$

where
$$\begin{aligned}
E' &\equiv \lambda e.\lambda m.m\,(\lambda x.x)\\
&\qquad (\lambda mn.(e\, m)(e\, n))\\
&\qquad (\lambda m.\lambda v.e(m\, v))\\
Y &\equiv \lambda h.(\lambda x.h(x\, x))(\lambda x.h(x\, x)).
\end{aligned}$$

Proof
We start by observing that

$$E\lceil M\rceil \rightarrow E''\lceil M\rceil \quad \text{and} \quad E''\lceil M\rceil \twoheadrightarrow \lceil M\rceil\, E_1\, E_2\, E_3$$

where
$$E'' \equiv (\lambda x.E'(x\,x))(\lambda x.E'(x\,x))$$
$$E_1 \equiv \lambda x.x$$
$$E_2 \equiv \lambda mn.(E''m)(E''n)$$
$$E_3 \equiv \lambda m.\lambda v.E''(m\,v).$$

We conclude by proving that
$$E''\lceil M\rceil \twoheadrightarrow M.$$

The proof is by induction over the structure of M. We assume that the property holds for all proper substructures of a term M, and conclude that it also holds for M. We look at the cases where M is a variable, an application or an abstraction

$M \equiv x, \quad x \in V$

$$\begin{aligned}
E''\lceil M\rceil &\twoheadrightarrow \lceil M\rceil E_1 E_2 E_3 \\
&\equiv (\lambda abc.a\,x) E_1 E_2 E_3 \quad \text{by representation of variables} \\
&\twoheadrightarrow E_1 x \\
&\equiv (\lambda x.x)\,x \\
&\rightarrow x \\
&\equiv M
\end{aligned}$$

$M \equiv M_1 M_2$

$$\begin{aligned}
E''\lceil M\rceil &\twoheadrightarrow \lceil M\rceil E_1 E_2 E_3 \\
&\equiv (\lambda abc.b\lceil M_1\rceil\lceil M_2\rceil) E_1 E_2 E_3 \quad \text{by representation of applications} \\
&\twoheadrightarrow E_2\lceil M_1\rceil\lceil M_2\rceil \\
&\equiv (\lambda mn.(E''m)(E''n))\lceil M_1\rceil\lceil M_2\rceil \\
&\twoheadrightarrow (E''\lceil M_1\rceil)(E''\lceil M_2\rceil) \\
&\twoheadrightarrow M_1 M_2 \quad \text{by induction assumption (twice)} \\
&\equiv M
\end{aligned}$$

$M \equiv \lambda x.M_1$

$$\begin{aligned}
E''\lceil M\rceil &\twoheadrightarrow \lceil M\rceil E_1 E_2 E_3 \\
&\equiv (\lambda abc.c(\lambda x.\lceil M_1\rceil)) E_1 E_2 E_3 \quad \text{by representation of abstractions} \\
&\twoheadrightarrow E_3(\lambda x.\lceil M_1\rceil) \\
&\equiv (\lambda m.\lambda v.E''(m\,v))(\lambda x.\lceil M_1\rceil) \quad \text{where } v \text{ does not occur free in } M_1 \\
&\rightarrow \lambda v.E''((\lambda x.\lceil M_1\rceil)\,v) \\
&\equiv \lambda x.E''((\lambda x.\lceil M_1\rceil)\,x) \quad \text{by } \alpha\text{-conversion} \\
&\rightarrow \lambda x.E''\lceil M_1\rceil \\
&\twoheadrightarrow \lambda x.M_1 \quad \text{by induction assumption} \\
&\equiv M.
\end{aligned}$$

The only non-obvious step is the use of α-conversion in the last case.

Appendix B: Outline of Proof of correctness for the self-reducer

We will start by proving that the self-reducer yields the representation of the normal form of its argument, when the argument has a normal form. Then we show that there is no normal form for the reducer applied to a term with no normal form.

For reasons of space, the proofs of some lemmas have been omitted. The proof of lemma 1 has been kept in full as an example of the techniques used. A complete proof can be obtained by contacting the author.

When there is a normal form

We assume that the term $M \in \Lambda^0$ reduces to a normal form NF_M. We then show that

$$R\lceil M\rceil =_\beta \lceil \mathsf{NF}_M\rceil$$

where

$$\begin{aligned}
R &\equiv \lambda m . R' m(\lambda ab . b)\\
R' &\equiv Y\lambda r . \lambda m . mR_1 R_2 R_3\\
R_1 &\equiv (\lambda x . x)\\
R_2 &\equiv (\lambda mn . (r\,m)\,(\lambda ab . a)\,(r\,n))\\
R_3 &\equiv \lambda m . R_3'(\lambda v . r(m\,v))\\
R_3' &\equiv \lambda g . \lambda x . x\,g(\lambda abc . c(\lambda w . g(P\,\lambda abc . a\,w)\,(\lambda ab . b)))\\
P &\equiv Y P'\\
P' &\equiv \lambda p . \lambda m . (\lambda x . x(\lambda v . p(\lambda abc . b\,m(v(\lambda ab . b)))) \,m)\\
Y &\equiv \lambda h . (\lambda x . h(x\,x))\,(\lambda x . h(x\,x)).
\end{aligned}$$

We first show a few lemmas and corollaries.

Lemma 1
For all λ-terms $M, N,$ and variables x

$$R'(\lceil M\rceil[x := R'\lceil N\rceil]) =_\beta R'\lceil M[x := N]\rceil.$$

Proof
We do induction over the structure of M

$M = x$

$$\begin{aligned}
& R'(\lceil M\rceil[x := R'\lceil N\rceil]) &&\\
\equiv\ & R'((\lambda abc . a\,x)[x := R'\lceil N\rceil]) && \text{representation of } x\\
\equiv\ & R'(\lambda abc . a(R'\lceil N\rceil)) && \text{substitution}\\
=_\beta\ & R_1(R'\lceil N\rceil) && \text{reduction}\\
=_\beta\ & R'\lceil N\rceil && \text{reduction}\\
\equiv\ & R'\lceil x[x := N]\rceil && \text{inverse substitution.}
\end{aligned}$$

$M = y, \quad y \neq x$

$$\begin{array}{lll}
 & R'(\lceil M \rceil [x := R'\lceil N \rceil]) & \\
\equiv & R'((\lambda abc.\,a\,y)[x := R'\lceil N \rceil]) & \text{representation of } y \\
\equiv & R'(\lambda abc.\,a\,y) & \text{substitution} \\
\equiv & R'\lceil y \rceil & \text{representation of } y \\
\equiv & R'\lceil y[x := N] \rceil & \text{inverse substitution}
\end{array}$$

$M = M_1 M_2$

$$\begin{array}{lll}
 & R'(\lceil M \rceil [x := R'\lceil N \rceil]) & \\
\equiv & R'((\lambda abc.\,b\lceil M_1 \rceil \lceil M_2 \rceil)[x := R'\lceil N \rceil]) & \text{representation of } M_1 M_2 \\
\equiv & R'(\lambda abc.\,b\lceil M_1 \rceil [x := R'\lceil N \rceil] \lceil M_2 \rceil [x := R'\lceil N \rceil]) & \\
=_\beta & R_2[r := R'] \lceil M_1 \rceil [x := R'\lceil N \rceil] \lceil M_2 \rceil [x := R'\lceil N \rceil] & \text{reduction} \\
=_\beta & (R'\lceil M_1 \rceil [x := R'\lceil N \rceil])(\lambda ab.\,a)(R'\lceil M_2 \rceil [x := R'\lceil N \rceil]) & \text{reduction} \\
=_\beta & (R'\lceil M_1[x := N] \rceil)(\lambda ab.\,a)(R'\lceil M_2[x := N] \rceil) & \text{induction (twice)} \\
=_\beta & R_2[r := R'] \lceil M_1[x := N] \rceil \lceil M_2[x := N] \rceil & \text{inverse reduction} \\
=_\beta & R'(\lambda abc.\,b\lceil M_1[x := N] \rceil \lceil M_2[x := N] \rceil) & \text{inverse reduction} \\
=_\beta & R'\lceil (M_1 M_2)[x := N] \rceil) & \text{representation of } M_1 M_2 \\
=_\beta & (R'\lceil M[x := N] \rceil) &
\end{array}$$

$M = \lambda y.\,M_1$

$$\begin{array}{lll}
 & R'(\lceil M \rceil [x := R'\lceil N \rceil]) & \\
\equiv & R'((\lambda abc.\,c(\lambda y.\lceil M_1 \rceil))[x := R'\lceil N \rceil]) & \text{representation of } \lambda y.\,M_1 \\
\equiv & R'(\lambda abc.\,c(\lambda y.\lceil M_1 \rceil [x := R'\lceil N \rceil])) & \\
=_\beta & R_3[r := R'](\lambda y.\lceil M_1 \rceil [x := R'\lceil N \rceil]) & \text{reduction} \\
=_\beta & (\lambda m.\,R_3'(\lambda v.\,R'(m\,v)))(\lambda y.\lceil M_1 \rceil [x := R'\lceil N \rceil]) & \text{substitution} \\
=_\beta & R_3'(\lambda v.\,R'((\lambda y.\lceil M_1 \rceil [x := R'\lceil N \rceil])\,v)) & \text{reduction} \\
=_\beta & R_3'(\lambda y.\,R'((\lambda y.\lceil M_1 \rceil [x := R'\lceil N \rceil])\,y)) & \alpha\text{-conversion} \\
=_\beta & R_3'(\lambda y.\,R'\lceil M_1 \rceil [x := R'\lceil N \rceil]) & \text{reduction} \\
=_\beta & R_3'(\lambda y.\,R'\lceil M_1[x := N] \rceil) & \text{induction} \\
=_\beta & R_3'(\lambda y.\,R'((\lambda y.\lceil M_1[x := N] \rceil)\,y)) & \text{inverse reduction} \\
=_\beta & R_3'(\lambda v.\,R'((\lambda y.\lceil M_1[x := N] \rceil)\,v)) & \alpha\text{-conversion} \\
=_\beta & (\lambda m.\,R_3'(\lambda v.\,R'(m\,v)))(\lambda y.\lceil M_1[x := N] \rceil) & \text{inverse reduction} \\
=_\beta & R_3[r := R'](\lambda y.\lceil M_1[x := N] \rceil) & \text{inverse substitution} \\
=_\beta & R'(\lambda abc.\,\lambda y.\lceil M_1[x := N] \rceil)) & \text{inverse reduction} \\
=_\beta & R'\lceil (\lambda y.\,M_1)[x := N] \rceil & \text{representation of } \lambda y.\,M_1 \\
=_\beta & R'\lceil M[x := N] \rceil. &
\end{array}$$

The only non-trivial parts are the use of α-conversion and inverse reduction. The latter can be seen as starting from both ends and working towards the middle using forwards reduction only. We use $=_\beta$ rather than $\twoheadrightarrow$ when we say 'reduction', as we in addition to plain reduction use the equivalence $YF =_\beta F(YF)$.

Lemma 1 has a few useful corollaries:

Corollary 1
For all λ-terms M, N, and variables y

$$R'\lceil(\lambda y\,.\,M)\,N\rceil =_\beta R'\lceil M[x := N]\rceil.$$

Proof

$$\begin{aligned}
& R'\lceil(\lambda y\,.\,M)\,N\rceil \\
=_\beta\;& R'\lceil \lambda y\,.\,M\rceil\,(\lambda ab\,.\,a)\,R'\lceil N\rceil && \text{reduction} \\
=_\beta\;& (R'_3(\lambda v\,.\,R'((\lambda y\,.\lceil M\rceil)\,v)))\,(\lambda ab\,.\,a)\,R'\lceil N\rceil && \text{reduction} \\
=_\beta\;& (R'_3(\lambda y\,.\,R'\lceil M\rceil))\,(\lambda ab\,.\,a)\,R'\lceil N\rceil && \alpha\text{-conversion and reduction} \\
=_\beta\;& (\lambda g\,.\,\lambda x\,.\,x\,g(\lambda ab\,c\,.\,c(\lambda w\,.\,g(P\,\lambda abc\,.\,a\,w)\,(\lambda ab\,.\,b)))) \\
& \quad (\lambda y\,.\,R'\lceil M\rceil)\,(\lambda ab\,.\,a)\,R'\lceil N\rceil && \text{substitution} \\
=_\beta\;& (\lambda y\,.\,R'\lceil M\rceil)\,R'\lceil N\rceil && \text{reduction} \\
\rightarrow\;& R'\lceil M\rceil\,[y := R'\lceil N\rceil] \\
=_\beta\;& R'\lceil M[y := N]\rceil && \text{Lemma 1.}
\end{aligned}$$

Corollary 2
For all λ-terms M, N

$$\text{if} \quad M \rightarrow N \quad \text{then} \quad R'\lceil M\rceil =_\beta R'\lceil N\rceil.$$

Proof
We omit the proof, which is a simple induction on the location of the redex.

Corollary 3
For all λ-terms M, N

$$\text{if} \quad M \twoheadrightarrow N \quad \text{then} \quad R'\lceil M\rceil =_\beta R'\lceil N\rceil.$$

Proof
This is a simple induction on the number of reduction steps. If zero, $M \equiv N$, if more than zero, we use Corollary 2 to reduce the number of reduction steps.

Corollary 4
For all λ-terms, M, N

$$\text{if} \quad M =_\beta N \quad \text{then} \quad R'\lceil M\rceil =_\beta R'\lceil N\rceil.$$

Proof
If $M =_\beta N$, then there exist a λ-term T, such that $M \twoheadrightarrow T$ and $N \twoheadrightarrow T$. By Corollary 3, $R'\lceil M\rceil =_\beta R'\lceil T\rceil =_\beta R'\lceil N\rceil$.

We have now proven that R', and thus R respects equivalence. We now need to show that R will return its argument unchanged if it is in normal form.

Lemma 2
For all λ-terms M, there exist a λ-term M', such that

$$P\lceil M\rceil =_{\beta} \lambda x\,.\,x\,M'\lceil M\rceil .$$

Proof

$$\begin{aligned} &P\lceil M\rceil \\ &=_{\beta} (\lambda x\,.\,x(\lambda v\,.\,P(\lambda abc\,.\,b\lceil M\rceil(v(\lambda ab\,.\,b))))\lceil M\rceil) \\ &=_{\beta} (\lambda x\,.\,x\,M'\lceil M\rceil) \end{aligned}$$

where $M' \equiv (\lambda v\,.\,P(\lambda abc\,.\,b\lceil M\rceil(v(\lambda ab\,.\,b))))$.

Lemma 3
For all λ-terms M, N, and N'

$$P\lceil M\rceil(\lambda ab\,.\,b)(\lambda x\,.\,x\,N'\lceil N\rceil) =_{\beta} P\lceil M\,N\rceil .$$

Proof

$$\begin{aligned} &P\lceil M\rceil(\lambda ab\,.\,a)(\lambda x\,.\,x\,N'\lceil N\rceil) \\ &=_{\beta} (\lambda x\,.\,x(\lambda v\,.\,P(\lambda abc\,.\,b\lceil M\rceil(v(\lambda ab\,.\,b))))\lceil M\rceil)(\lambda ab\,.\,a)(\lambda x\,.\,x\,N'\lceil N\rceil) \\ &=_{\beta} (\lambda v\,.\,P(\lambda abc\,.\,b\lceil M\rceil(v(\lambda ab\,.\,b))))(\lambda x\,.\,x\,N'\lceil N\rceil) \\ &=_{\beta} P(\lambda abc\,.\,b\lceil M\rceil((\lambda x\,.\,x\,N'\lceil N\rceil)(\lambda ab\,.\,b))) \\ &=_{\beta} P(\lambda abc\,.\,c\lceil M\rceil\lceil N\rceil) \\ &=_{\beta} P\lceil M\,N\rceil . \end{aligned}$$

Lemma 4
Given a normal form λ-term $M \in \mathsf{NF}_{\Lambda}$, where $\mathsf{FV}(M) = \{z_1, \ldots, z_n\}$, if $M \equiv \lambda y\,.\,M_1$ then there exists a λ-term M' such that

$$R'(\lceil M\rceil[z_i := P\lceil z_i\rceil]) =_{\beta} \lambda x\,.\,x\,M'\lceil M\rceil$$

otherwise $R'(\lceil M\rceil[z_i := P\lceil z_i\rceil]) =_{\beta} P\lceil M\rceil$

where we use the notation $[z_i := P\lceil z_i\rceil]$ as an abbreviation of $[z_1 := P\lceil z_1\rceil, \ldots, z_n := P\lceil z_n\rceil]$. Note that by Lemma 2, $P\lceil M\rceil =_{\beta} \lambda x\,.\,x\,M'\lceil M\rceil$ for some M'.

Proof
We do induction over the structure of M. Details omitted.

Now we finally get to

Theorem 2
Given a closed λ-term $M \in \Lambda^0$ with a normal form $\mathsf{NF}_M \in \mathsf{NF}_{\Lambda}$, we have

$$R\lceil M\rceil =_{\beta} \lceil \mathsf{NF}_M\rceil .$$

14-2

Proof

$R\lceil M\rceil$	
$=_\beta R'\lceil M\rceil(\lambda ab.b)$	reduction
$=_\beta R'\lceil \mathsf{NF}_M\rceil(\lambda ab.b)$	Corollary 3
$=_\beta(\lambda x.x\,M'\lceil \mathsf{NF}_M\rceil)(\lambda ab.b)$	Lemmas 4 and 2
$=_\beta\lceil \mathsf{NF}_M\rceil)$	reduction.

When there is no normal form

We will start by presenting two functions $\mathbb{R}[\![\;]\!]$ and $\mathbb{R}'[\![\;]\!]$ that map λ-terms to λ-terms. Then we show that $R\lceil M\rceil =_\beta \mathbb{R}[\![M]\!][\,]$, then that if $M \to N$ then $\mathbb{R}[\![M]\!]\rho \twoheadrightarrow \mathbb{R}[\![N]\!]\rho$ using at least one left-most reduction step. Finally, we use this to conclude that if there is an infinite left-most reduction path for M, then there is an infinite quasi-left most reduction path for $\mathbb{R}[\![M]\!][\,]$, and thus we see that if M has no normal form, then neither has $R\lceil M\rceil$. We will use $\underset{l}{\to}$ to describe a single left-most reduction step.

We first define

$$P'' \equiv (\lambda x.P'(x\,x))(\lambda x.P'(x\,x))$$
$$P''' \equiv \lambda m.(\lambda x.x(\lambda v.P''(\lambda abc.b\,m(v(\lambda ab.b))))\,m)$$

and note that $P \twoheadrightarrow P''$ and $P'' \twoheadrightarrow P'''$.

In the mappings $\mathbb{R}[\![\;]\!]$ and $\mathbb{R}'[\![\;]\!]$ we use environments ρ, ρ', etc. We use $[\,]$ to signify the empty environment and use $\rho[x_1 \mapsto M_1, \ldots, x_n \mapsto M_n]$ as the environment ρ extended by binding x_1 to M_1, etc. Environments will always bind variables either to themselves or to their representations

$$\mathbb{R}[\![x\,M_1 \ldots M_n]\!]\rho \equiv \lambda abc.b(\ldots(\lambda abc.b\lceil x\rceil\,\mathbb{R}[\![M_1]\!]\rho)\ldots)\,\mathbb{R}[\![M_n]\!]\rho \quad \text{if } n \geqslant 0, \rho\,x = \lceil x\rceil$$
$$\mathbb{R}[\![x\,M_1 \ldots M_n]\!]\rho \equiv \mathbb{R}'[\![x\,M_1 \ldots M_n]\!]\rho(\lambda ab.b) \quad \text{if } n \geqslant 0, \rho\,x = x$$
$$\mathbb{R}[\![(\lambda x.M_0)\,M_1 \ldots M_n]\!]\rho \equiv \mathbb{R}'[\![(\lambda x.M_0)\,M_1 \ldots M_n]\!]\rho(\lambda ab.b) \quad \text{if } n \geqslant 1$$
$$\mathbb{R}[\![\lambda x.M_0]\!]\rho \equiv \lambda abc.c(\lambda x.\mathbb{R}[\![M_0]\!]\rho[x \mapsto \lceil x\rceil])$$
$$\mathbb{R}'[\![x\,M_1 \ldots M_n]\!]\rho \equiv P''(\lambda abc.b(\ldots(\lambda abc.b\lceil x\rceil\,\mathbb{R}[\![M_1]\!]\rho)\ldots)\,\mathbb{R}[\![M_n]\!]\rho) \qquad \text{if } n \geqslant 0, \rho\,x = \lceil x\rceil$$
$$\mathbb{R}'[\![x\,M_1 \ldots M_n]\!]\rho \equiv x(\lambda ab.a)\,\mathbb{R}'[\![M_1]\!]\rho \ldots (\lambda ab.a)\,\mathbb{R}'[\![M_n]\!]\rho \quad \text{if } n \geqslant 0, \rho\,x = x$$
$$\mathbb{R}'[\![(\lambda x.M_0)\,M_1 \ldots M_n]\!]\rho \equiv (\lambda x.\mathbb{R}'[\![M_0]\!]\rho[x \mapsto x])\,\mathbb{R}'[\![M_1]\!]\rho \ldots (\lambda ab.a)\,\mathbb{R}'[\![M_n]\!]\rho \qquad \text{if } n \geqslant 1$$
$$\mathbb{R}'[\![\lambda x.M_0]\!]\rho \equiv \lambda z.z(\lambda x.\mathbb{R}'[\![M_0]\!]\rho[x \mapsto x])(\lambda abc.c(\lambda x.\mathbb{R}[\![M_0]\!]\rho[x \mapsto \lceil x\rceil])).$$

Lemma 5
If M is in normal form and $\rho\,x = \lceil x\rceil$ for all free variables in M, then $\mathbb{R}[\![M]\!]\rho \equiv \lceil M\rceil$.

Proof
If the conditions are true, only the first and last rule of $\mathbb{R}[\![\;]\!]$ are used. It is easy to see that they build the representations of their arguments. Note that when applying $\mathbb{R}[\![\;]\!]$ under a lambda, the environment is updated in such a fashion as to preserve the conditions.

Lemma 6
For all λ-terms M and all environments ρ

$$\mathbb{R}'[\![M]\!]\,\rho(\lambda ab.b) \twoheadrightarrow \mathbb{R}[\![M]\!]\,\rho.$$

Proof
All but the first rule of $\mathbb{R}'[\![\ ]\!]$ give the result trivially. It is easy to see that $P''M \twoheadrightarrow \lambda x.x M' M$ for some M'. Thus $P''M(\lambda ab.b) \twoheadrightarrow M$, which when applied to the first rule of $\mathbb{R}'[\![\ ]\!]$ give the desired result.

Lemma 7
For all λ-terms M with free variables v_i and v_j

$$R'\lceil M\rceil\,[v_i := P\lceil v_i\rceil] =_\beta \mathbb{R}'[\![M]\!]\,[v_i \mapsto \lceil v_i\rceil,\quad v_j \mapsto v_j].$$

Proof
We do induction over the structure of M. Details omitted.

Corollary 5
For all λ-terms M with free variables v_i and v_j

$$R\lceil M\rceil\,[v_i := P\lceil v_i\rceil] =_\beta \mathbb{R}[\![M]\!]\,[v_i \mapsto \lceil v_i\rceil, v_j \mapsto v_j].$$

Proof

$$\begin{array}{lll} & R\lceil M\rceil\,[v_i := P\lceil v_i\rceil] & \\ \to & R'\lceil M\rceil\,[v_i := P\lceil v_i\rceil]\,(\lambda ab.b) & \\ =_\beta & \mathbb{R}'[\![M]\!]\,[v_i := P\lceil v_i\rceil, v_j \mapsto v_j]\,(\lambda ab.b) & \text{Lemma 7} \\ \twoheadrightarrow & \mathbb{R}[\![M]\!]\,[v_i := P\lceil v_i\rceil, v_j \mapsto v_j] & \text{Lemma 6.} \end{array}$$

Lemma 8
For all λ-terms M and N

$$\mathbb{R}'[\![M]\!]\,\rho(\lambda ab.a)\,\mathbb{R}'[\![N]\!]\,\rho \twoheadrightarrow \mathbb{R}'[\![M\,N]\!]\,\rho.$$

Proof
We do induction over the structure of M. Details omitted.

We say that a variable is free in an environment ρ, if it is free in ρx for some x.

Lemma 9
For all λ-terms M and N, environments ρ and variables y not free in ρ

$$(\mathbb{R}'[\![M]\!]\,\rho[y \mapsto y])\,[y := \mathbb{R}'[\![N]\!]\,\rho] \twoheadrightarrow \mathbb{R}'[\![M[y := N]]\!]\,\rho$$

and
$$(\mathbb{R}[\![M]\!]\,\rho[y \mapsto y])\,[y := \mathbb{R}'[\![N]\!]\,\rho] \twoheadrightarrow \mathbb{R}[\![M[y := N]]\!]\,\rho.$$

Proof
We do induction over the structure of M. Details omitted.

We are now getting close to the end:

Lemma 10
For all λ-terms $M_0, \ldots, M_n$, $n \geqslant 1$ and all environments ρ

$$\mathbb{R}[\![(\lambda y . M_0)\, M_1\, M_2 \ldots M_n]\!]\, \rho \twoheadrightarrow \mathbb{R}[\![(M_0[y := M_1]\, M_2 \ldots M_n]\!]\, \rho$$

using at least one left-most reduction step.

Proof

$$\begin{aligned}
&\mathbb{R}[\![(\lambda y . M_0)\, M_1\, M_2 \ldots M_n]\!]\, \rho \\
&\equiv \mathbb{R}'[\![(\lambda y . M_0)\, M_1\, M_2 \ldots M_n]\!]\, \rho(\lambda ab . b) \\
&\equiv (\lambda y . \mathbb{R}'[\![M_0]\!]\, \rho[y \mapsto y])\, \mathbb{R}'[\![M_1]\!]\, \rho(\lambda ab . a)\, \mathbb{R}'[\![M_2]\!]\, \rho \\
&\qquad \ldots (\lambda ab . a)\, \mathbb{R}'[\![M_n]\!]\, \rho(\lambda ab . b) \\
&\underset{l}{\rightarrow} (\mathbb{R}'[\![M_0]\!]\, \rho[y \mapsto y])\, [y := \mathbb{R}'[\![M_1]\!]\, \rho]\, (\lambda ab . a)\, \mathbb{R}'[\![M_2]\!]\, \rho \\
&\qquad \ldots (\lambda ab . a)\, \mathbb{R}'[\![M_n]\!]\, \rho(\lambda ab . b) \\
&\twoheadrightarrow \mathbb{R}'[\![M_0[y := M_1]]\!]\, \rho(\lambda ab . a)\, \mathbb{R}'[\![M_2]\!]\, \rho \ldots (\lambda ab . a)\, \mathbb{R}'[\![M_n]\!]\, \rho(\lambda ab . b) && \text{Lemma 9} \\
&\twoheadrightarrow \mathbb{R}'[\![M_0[y := M_1]\, M_2 \ldots M_n]\!]\, \rho(\lambda ab . b) && \text{Lemma 8} \\
&\twoheadrightarrow \mathbb{R}[\![M_0[y := M_1]\, M_2 \ldots M_n]\!]\, \rho && \text{Lemma 6.}
\end{aligned}$$

We now generalize this to arbitrary redexes.

Lemma 11
For all λ-terms M and N where $M \underset{l}{\rightarrow} N$ and all environments ρ, where $\rho x = \lceil x \rceil$ for all variables $x \in \mathsf{FV}(M)$

$$\mathbb{R}[\![M]\!]\, \rho \twoheadrightarrow \mathbb{R}[\![N]\!]\, \rho$$

using at least one left-most reduction step.

Proof
We do induction on the position of the left-most redex

$$M = (\lambda x . M_0)\, M_1, \ldots, M_n, \quad n \geqslant 1.$$

We get the result directly from Lemma 10

$M = x\, M_1 \ldots M_i \ldots M_n, \quad n \geqslant 0,$ the left-most redex is in M_i, $M_i \underset{l}{\rightarrow} N_i$

$$\begin{aligned}
&\mathbb{R}[\![M = x\, M_1 \ldots M_i \ldots M_n]\!]\, \rho \\
&\equiv \lambda abc . b(\ldots (\lambda abc . b\lceil x \rceil\, \mathbb{R}[\![M_1]\!]\, \rho \ldots \mathbb{R}[\![M_i]\!]\, \rho \ldots)\, \mathbb{R}[\![M_n]\!]\, \rho \\
&\twoheadrightarrow \lambda abc . b(\ldots (\lambda abc . b\lceil x \rceil\, \mathbb{R}[\![M_1]\!]\, \rho \ldots \mathbb{R}[\![N_i]\!]\, \rho \ldots)\, \mathbb{R}[\![M_n]\!]\, \rho && \text{by induction} \\
&\equiv \mathbb{R}[\![M = x\, M_1 \ldots N_i \ldots M_n]\!]\, \rho.
\end{aligned}$$

In the induction step we use that for $j < i$, $\mathbb{R}[\![M_j]\!]\, \rho \equiv \lceil M_j \rceil$, since M_j is in normal form (Lemma 5). This means that $\mathbb{R}[\![M_j]\!]\, \rho$ contains no redexes, so the left-most redex of the entire expression must be the left-most redex of $\mathbb{R}[\![M_l]\!]\,\rho$. Hence we get the required left-most reduction step.

$M = \lambda x.M_0$

$$\begin{aligned} & \mathbb{R}[\![\lambda x.M_0]\!]\,\rho \\ \equiv\ & \lambda abc.c(\lambda x.\mathbb{R}[\![M_0]\!]\,\rho[x \mapsto [x]]) \\ \twoheadrightarrow\ & \lambda abc.c(\lambda x.\mathbb{R}[\![N_0]\!]\,\rho[x \mapsto [x]]) \quad \text{by induction} \\ \equiv\ & \mathbb{R}[\![\lambda x.N_0]\!]\,\rho. \end{aligned}$$

Again, we note that the left-most redex in $\mathbb{R}[\![M_0]\!]\,\rho[x \mapsto [x]]$ is the left-most redex in the entire expression.

Now we are ready for the finale:

Theorem 3
If a closed λ-term M has no normal form, then $R\lceil M \rceil$ has no normal form either.

Proof
If M has no normal form, then the sequence of left-most reduction steps is infinite

$$M \underset{l}{\rightarrow} M_1 \underset{l}{\rightarrow} M_2 \underset{l}{\rightarrow} \dots .$$

By Lemma 11 we have

$$\mathbb{R}[\![M]\!]\,[\,] \twoheadrightarrow \mathbb{R}[\![M_1]\!]\,[\,] \twoheadrightarrow \mathbb{R}[\![M_2]\!]\,[\,] \twoheadrightarrow \dots$$

where each of the $\twoheadrightarrow$ contains at least one left-most reduction step. As we have only finitely many reductions between each left-most reduction step, we have an infinite quasi-left-most reduction sequence for $\mathbb{R}[\![M]\!]\,[\,]$, so by Theorem 13.2.6 in Barendregt (1984) it has no normal form. Since $R\lceil M \rceil =_\beta \mathbb{R}[\![M]\!]\,[\,]$ by Lemma 5, it has no normal form either.

References

Barendregt, H. 1984. *The Lambda Calculus: Its Syntax and Semantics* (*revised edition*). North-Holland.

Barendregt, H. 1991. Self-interpretation in lambda calculus. *J. Functional Program.*, **1** (2): 229–33.

Berger, U. and Schwichtenberg, H. 1991. An Inverse of the Evaluation Functional for Typed λ-calculus. *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, 203–211.

Dybvig, R. K. 1987. *The Scheme Programming Language*. Prentice-Hall.

Gomard, C. and Jones, N. D. 1991. A Partial Evaluator for the Untyped Lambda Calculus. *J. Functional Program.*, **1** (1): 21–69.

Kleene, S. C. 1952. *Introduction to Metamathematics*. North-Holland.

Pfenning, F. and Elliot, C. 1988. Higher-Order Abstract Syntax. In *Proc. ACM-SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 199–208. ACM Press.

Pfenning, F. and Lee, P. 1991. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, **89**: 137–159.

Reynolds, J. C. 1985. Three Approaches to Type Structure. Vol. 85 of *Lecture Notes in Computer Science*, pp. 97–138. Springer-Verlag.

Steensgaard-Madsen, J. 1989. Typed Representation of Objects by Functions. *TOPLAS*, **11** (1): 67–89.